Politechnika Warszawska

WYDZIAŁ ELEKTRYCZNY

Instytut Elektrotechniki Teoretycznej i Systemów Informacyjno-Pomiarowych

Praca dyplomowa magisterska

na kierunku Informatyka Stosowana

w specjalności Inżynieria Oprogramowania

Porównanie efektywności wybranych narzędzi służących do serwowania danych

inż. Jan Łukomski

numer albumu 291089

promotor

prof. dr hab. inż. Remigiusz Rak

WARSZAWA 2024

Porównanie efektywności wybranych narzędzi służących do serwowania danych

Streszczenie

W dzisiejszym świecie aplikacji internetowych, efektywne serwowanie danych jest kluczowym elementem zapewnienia komfortu użytkownika. W celu porównania wydajności różnych narzędzi służących do tego celu, przeprowadzono badania, których wyniki są prezentowane w niniejszej pracy.

Analiza koncentruje się na trzech popularnych frameworkach: Django, .NET oraz NestJS. Brane pod uwagę są różne czynniki, między innymi czas odpowiedzi oraz wydajność w zróżnicowanych warunkach działania. Celem badania jest nie tylko ocena samej szybkości uzyskiwania danych z bazy w ramach pojedynczego zapytania, ale także ocena zdolności frameworków do obsługi wielu użytkowników oraz większych zbiorów danych.

Podczas badań szczególną uwagę zwrócono na specyficzne cechy każdego narzędzia oraz jego potencjalne zalety i ograniczenia. Praca ma na celu dostarczenie obiektywnych danych, które mogą wspomóc programistów i inżynierów oprogramowania w dokonywaniu świadomych wyborów technologicznych podczas projektowania i wdrażania aplikacji internetowych.

Wyniki prezentowane w pracy stanowią istotne źródło informacji dla osób zainteresowanych optymalizacją wydajności aplikacji internetowych oraz wyborem odpowiedniego narzędzia do konkretnych projektów. Dalsze badania w tej dziedzinie mogą prowadzić do lepszego zrozumienia różnic między poszczególnymi frameworkami oraz doskonalenia technik serwowania danych w aplikacjach internetowych.

Słowa kluczowe: .NET, K6, Django, Docker, Testy obciążeniowe

Comparison of the effectiveness of selected data serving tools

Abstract

In today's world of web applications, serving data efficiently is a key element in ensuring a satisfactory user experience. In order to understand and compare the performance of various tools for this purpose, research was conducted, the results of which are presented in this work.

The analysis focuses on three popular frameworks: Django, .NET and NestJS. Various factors are taken into account, including response time and performance under various operating conditions. The aim of the study is not only to assess the speed of obtaining data from the database in a single query, but also to examine the framework's ability to handle multiple users and larger data.

During the research, special attention was paid to the specific features of each tool and their potential advantages and limitations. The work aims to provide objective data that can assist developers and software engineers in making informed technology choices when designing and implementing web applications.

The results presented in this article are an important source of information for people interested in optimizing the performance of web applications and choosing the right tool for specific projects. Further research in this area may lead to an even better understanding of the differences between individual frameworks and improvement of data serving techniques in web applications.

Keywords: .NET, K6, Django, Docker, Load tests

Spis treści

Rozdział 1

Wstęp

Wybór technologii w ramach projektu jest decyzją strategiczną, której nie jest łatwo zmienić w późniejszym etapie. W celu utrzymywania produktu warto czasami podjąć trudną decyzję o zmodernizowaniu stosu technologicznego. Specyfika warunków oraz okoliczności tworzenia oprogramowania są kluczowymi elementami mającym bardzo duży wpływ na wybór technologii.

1.1 Cel i zakres pracy

W pracy dokonano analizy porównawczej trzech wybranych frameworków serwujących dane: Django, .NET oraz NestJS. Pod uwagę został wzięty aspekt czasu odpowiedzi w różnych warunkach. Badanie ma na celu wskazać mocne oraz słabe strony każdego z porównywanych narzędzi. Badany jest wycinek rzeczywistości, który najlepiej jest przedstawiać poprzez analizę porównawczą. Bezwzględne wartości mogą się różnić w zależności od warunków uruchomienia, natomiast relacje między badanymi obiektami powinny być stale zauważalne.

Testy można podzielić na rodzaje pod różnymi względami. Podstawowy podział testów został zaprezentowany na rysunku 1[4]. Testy jednostkowe są pisane na stosunkowo niskim poziomie. Nie jest jednoznacznie zdefiniowane czym jest jednostka która jest testowana. Czasami jest to jedna funkcja, czasem jeden moduł. Wszystko zależy od kontekstu badanego rozwiązania. Różnią się szybkością działania. Nie jest potrzebna rozbudowana infrastruktura. Dzięki temu można je uruchamiać często.

Kolejnymi testami, realizowanymi na wyższym poziomie są testy integracyjne. Służą one badaniu współdziałania różnych modułów ze sobą. Do ich uruchomienia potrzebne jest zestawienie kilku modułów ze sobą więc wymagają one zazwyczaj większej infrastruktury niż testy jednostkowe. Wiąże się to z ograniczeniami związanymi ze zwiększonymi kosztami czasowymi oraz często, powiązanymi z tym, kosztami finansowymi.

Testy funkcjonalne nastawione są na sprawdzenie wymagań biznesowych aplikacji. Ich złożoność podobna jest do testów integracyjnych. Dziedziną badania jest sprawdzenie czy kluczowe elementy ich funkcjonalności są realizowane.

Kompleksowe testy służą do badania całej aplikacji. Testy te wymagają uruchomienia całego systemu w celu jego przetestowania. Z powodu ich kosztowności zaleca się ograniczenie liczby tych testów do minimum. Uruchomienie ich najlepiej weryfikuje czy badana aplikacja działa.

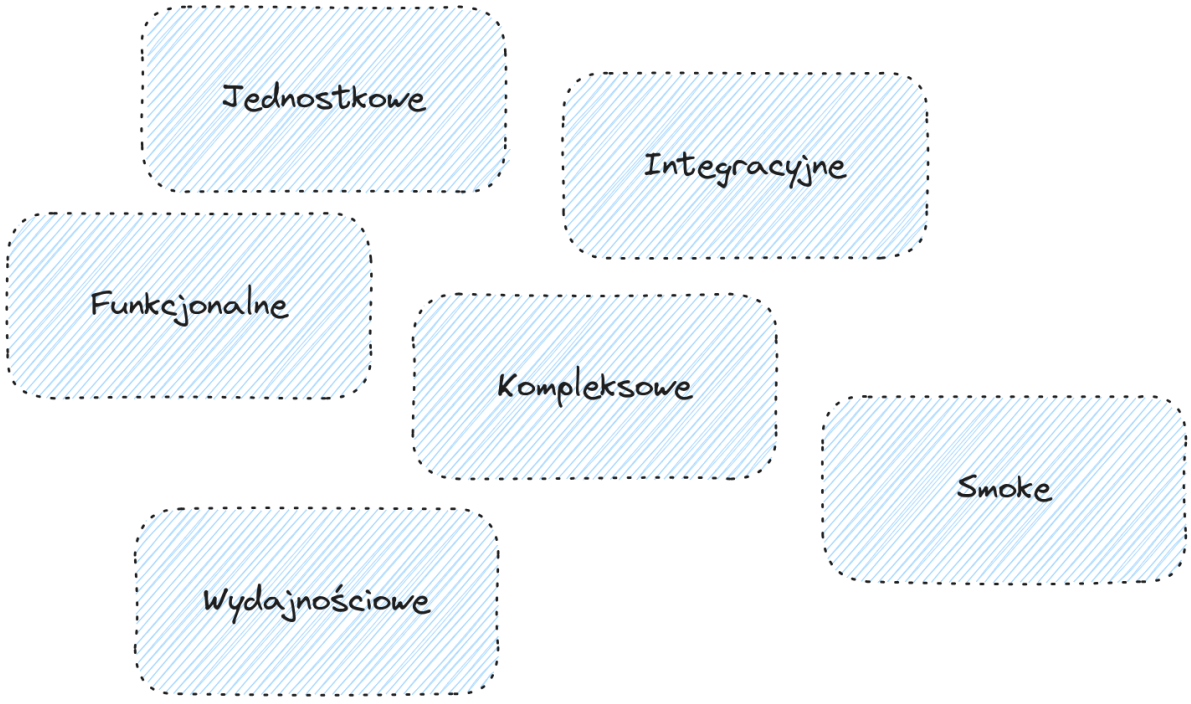
Testy akceptacyjne służą do weryfikacji czy kluczowe wymagania biznesowe są realizowane. Wymagają one również uruchomienia całej aplikacji przez co są kosztowne.

Smoke testy są rodzajem testów, które mają wyłapać błędy widoczne na pierwszy rzut oka. Polegają na pobieżnym przejściu przez dowolny fragment systemu. Jest to szybka weryfikacja czy podstawowy element systemu działa jak powinien.

Testy wydajnościowe służą sprawdzeniu czy w wymaganym obciążeniu aplikacja zachowuje się poprawnie. Pomagają znaleźć wąskie gardła systemu i zapobiec przerwie w działania w przypadku wzrostu obsługiwanych użytkowników lub zwiększenia liczby przetwarzanych danych,

Wraz z wchodzeniem na wyższy poziom testowania koszty testów rosną [11]. Stąd, istnieje piramida testów prezentująca zależność liczby testów w funkcji ich rodzaju. Piramida ta została zaprezentowana na rysunku 2. Testy jednostkowe są realizowane na najniższym poziomie piramidy, a więc tego rodzaju testów powinno być najwięcej jako, że są tanie w utrzymaniu. Wraz ze wzrostem poziomu piramidy, koszt utrzymania oraz uruchomienia testów rośnie. Sposób podziału testów jest umowny więc wszystkie nie wymienione rodzaje testów również znajdują się w piramidzie obok najbliższego im rodzaju testów.

Testy przeprowadzone w niniejszej pracy przynależą do kategorii testów wydajnościowych. Są to kluczowe testy zapewniające o jednym z istotnych aspektów dobrze działającego oprogramowania.



Rysunek 1. Rodzaje testów

Ich zadaniem jest identyfikacja obszarów ryzyka zachowania w czasie, wykorzystania zasobów oraz przepustowości [14].

Typy testów wydajnościowych zostały zaprezentowane na rysunku 3.

Testowanie obciążenia to badanie zdolności systemu do obsługi wzrastającego obciążenia przy realistycznych, przewidywanych liczbach użytkowników mających interakcje z systemem.

Testowanie przeciążające to badanie realizowane przy danych przekraczających realistyczne obciążenie. Ma ono na celu znalezienie granicy.

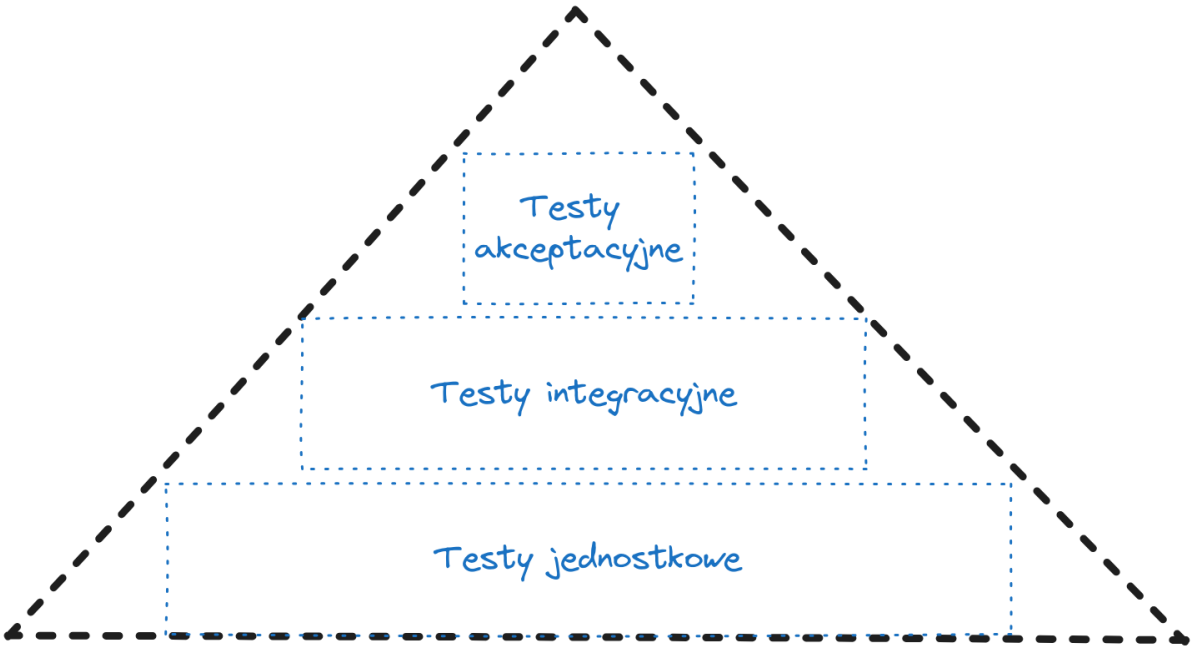
Testowanie skalowalności polega na zbadaniu granic funkcjonowania systemu przy zachowaniu większości pierwotnych założeń. Pozwala ono na monitorowanie i wczesne reagowanie gdy system wraz z rozwojem stopniowo przekracza bariery wytyczonych wymagań.

Testowanie skokowe to badanie reakcji systemu na nagły, chwilowy wzrost obciążenia. System powinien być w stanie wrócić no normalnego funkcjonowania, gdy obciążenie wróci do początkowego stanu.

Testowanie wytrzymałościowe skupia się na ocenie działania system w dłuższym okresie specyficznym dla rozwiązania. Chodzi o drobne uchybienia, które wraz z czasem działania mogą urosnąć do rangi poważnych błędów. Mogą to być np. błędy obliczeń związane z zaokrąglaniem lub wycieki pamięci.

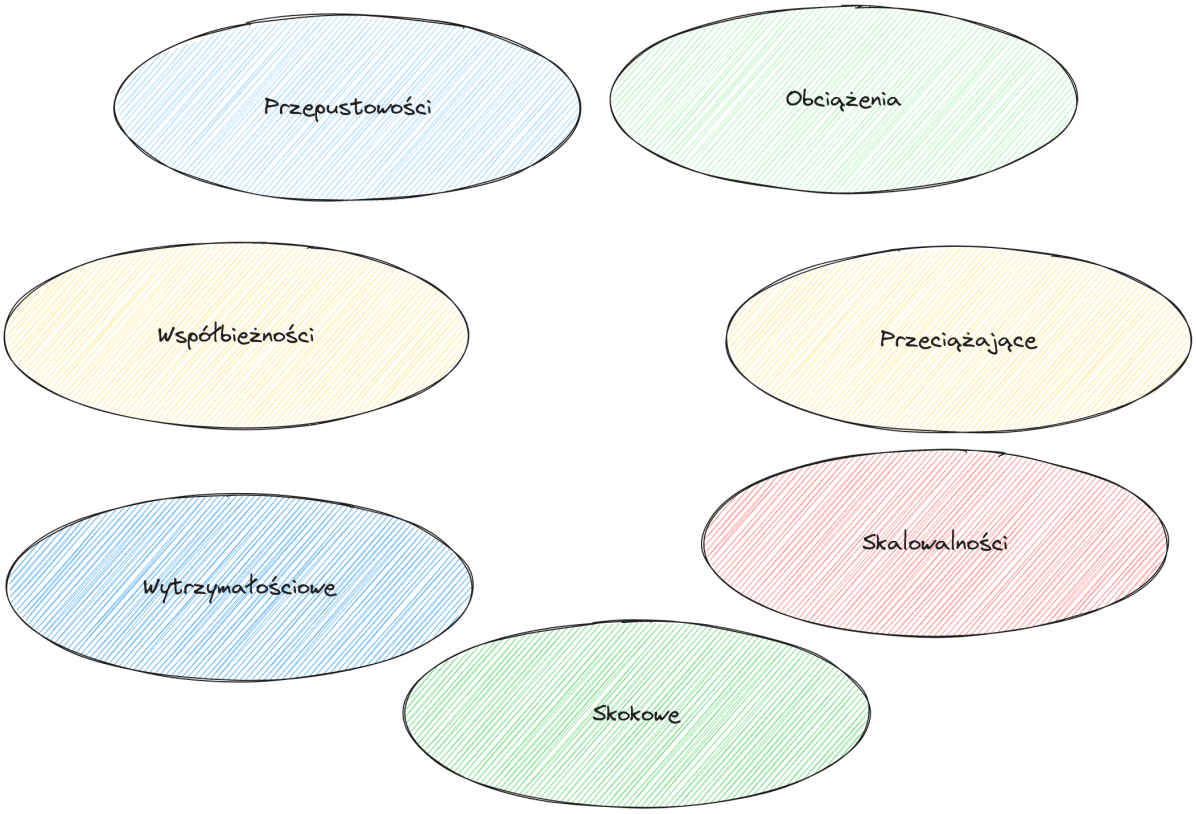
Testowanie współbieżności służy do sprawdzenia czy system działa poprawnie w przypadku przeprowadzania równoległych operacji. Jest to zazwyczaj trudny obszar do badania.

Testowanie przepustowości daje ocenę granic działania systemu zgodnie z wymaganiami przy zwiększonej liczbie aktorów mających interakcje z systemem.



Rysunek 2. Piramida testów

Z innego punktu widzenia testy można podzielić na statyczne oraz dynamiczne. Testowanie statyczne wiąże się ze sprawdzeniem wymagań, architektury rozwiązania w celu znalezienia wąskiego gardła. Czasami oznacza to również sprawdzenie używanych algorytmów. Jest to bardzo ważna weryfikacja, ponieważ może zostać zrobiona w początkowej fazie projektu. Poprawa problemu we wczesnej fazie zazwyczaj sprawia że jest on zdecydowanie mniej kosztowny. Testowanie dynamiczne to testy jednostkowe, integracyjne, akceptacyjne, które zazwyczaj powstają wraz z rozwojem systemu. Ich celem jest zwrócenie uwagi na problemy niedostrzeżone w fazie testowania statycznego.



Rysunek 3. Typy testów wydajnościowych

Rozdział 2

Przegląd literaturowy

2.1 Badanie Okami

Pracę o podobnej tematyce wykonał francuski programista Adrien Beaudouin na blogu Okami [2].

Autor przeprowadził analizę wydajności różnych frameworków webowych w kontekście obsługi bazy danych PostgresSQL. Zbadał finalne wyniki liczby żądań na sekundę dla każdego frameworka i porównał je, zwracając uwagę na korzyści i wady poszczególnych rozwiązań. Ponadto, dokonał oceny innych czynników, takich jak doświadczenie deweloperskie oraz wydajność w kontekście języków kompilowanych i interpretowanych. Praca zawiera również wzmianki o narzędziach ułatwiających konfigurację środowiska produkcyjnego. Na koniec, autor podsumował swoje wnioski, zwracając uwagę na istotę wyboru frameworka webowego nie tylko pod kątem wydajności, ale także doświadczenia deweloperskiego.

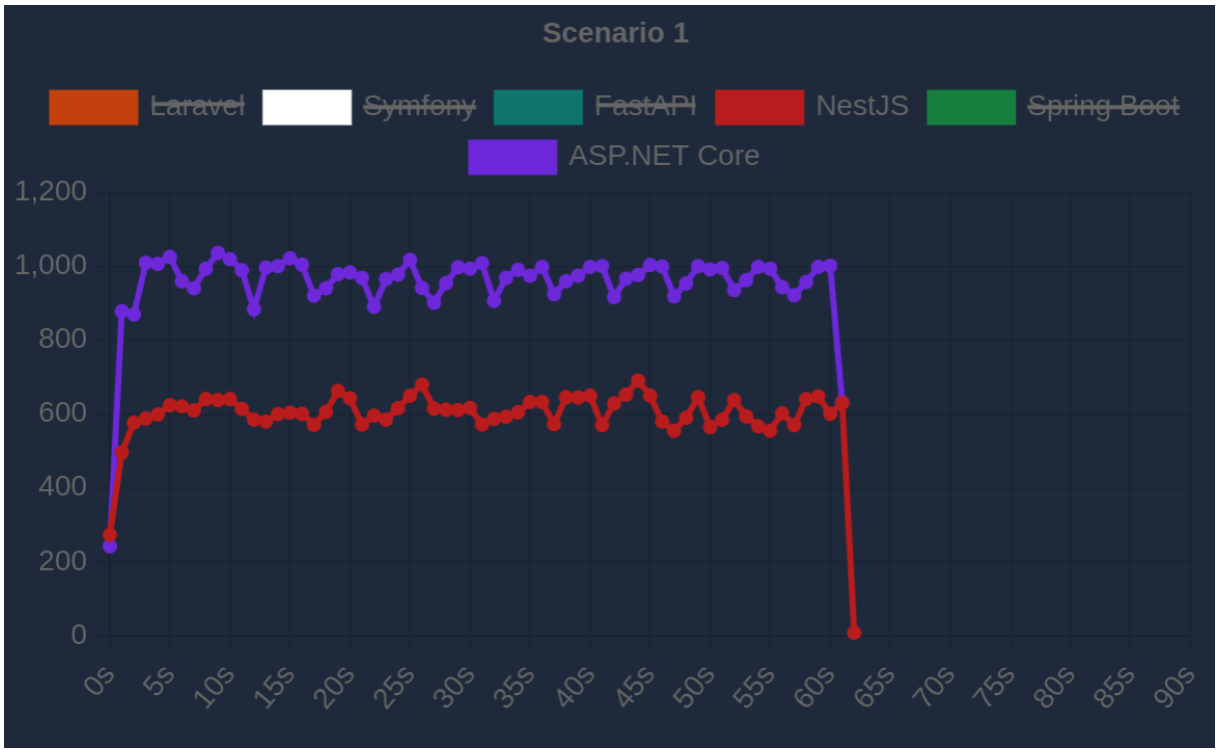
Badane były dwa scenariusze. W pierwszym nacisk położony był na wykorzystanie zasobów bazodanowych. Zapytania skonstruowane były tak, żeby potrzebne było wyciąganie zagnieżdżonych danych z bazy. Test ten miał sprawdzić jak sobie radzi aplikacja gdy styk bazy danych i frameworka jest obciążony. Rezultaty testów tego scenariusza zostały zaprezentowane na rysunku 4.

W drugim scenariuszu aplikacja była zasypywana dużą liczbą zapytań. Dane były wyciągane z bazy danych, ale były one stosunkowo proste i nie wymagały budowania większych zapytań. Nacisk został położony na samą obsługę dużej liczby zapytań przez framework. Wynik testów przedstawiony został na rysunku 5.

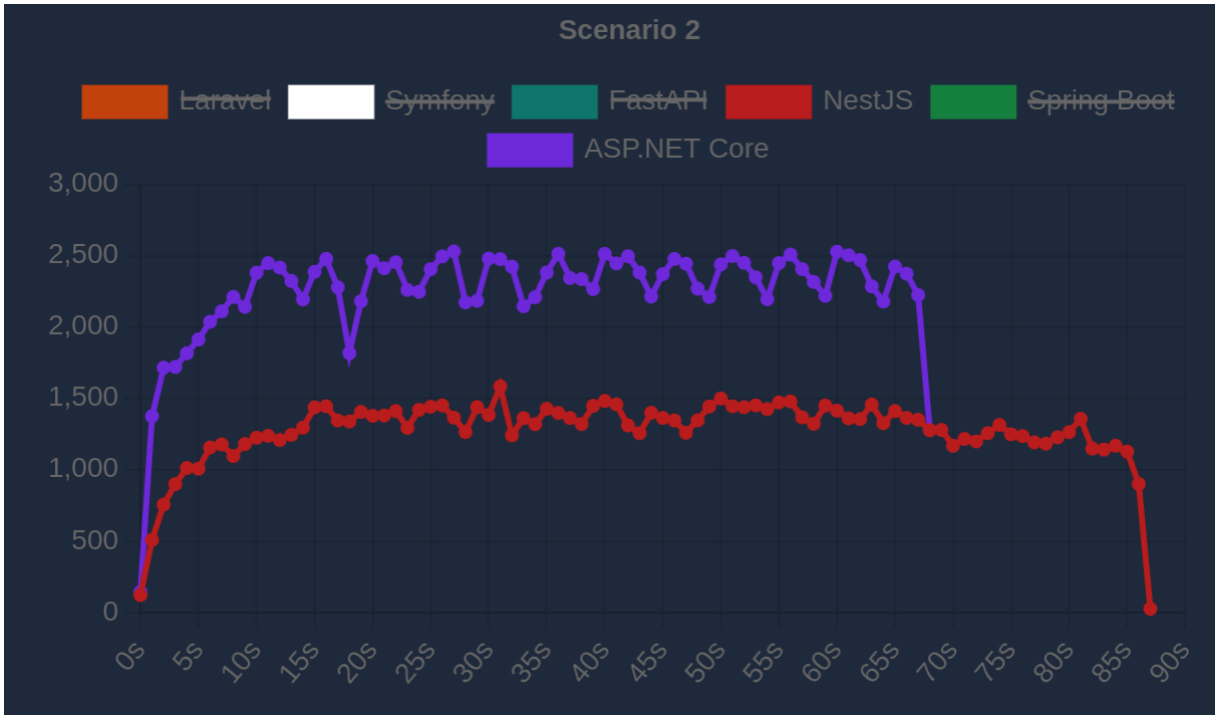
Wśród porównywanych frameworków znajdują się .NET oraz NestJS. Ten drugi konsekwentnie w obu testach wykazał się krótszym czasem odpowiedzi.

2.2 Ranking TechEmpower

TechEmpower to projekt Benchmarków Frameworków Webowych który jest otwartym źródłem [13]. Projekt ten mierzy wydajność wielu języków i frameworków aplikacji, co pozwala na ocenę wpływu wyboru technologii na wyniki wydajności. Benchmarki te są szeroko oglądane przez programistów



Rysunek 4. Zestawienie wyników testów w scenariuszu 1 - źródło: [2]

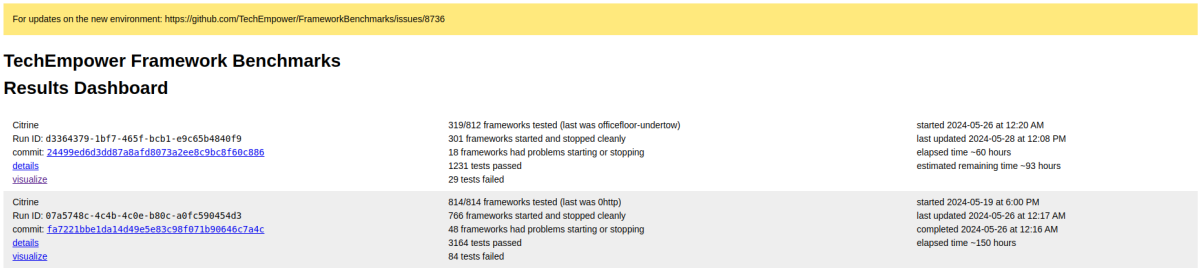


Rysunek 5. Zestawienie wyników testów w scenariuszu 2 - źródło: [2]

aplikacji internetowych i cieszą się dużym zainteresowaniem w branży. Celem TechEmpower jest dostarczenie obiektywnych danych na temat wydajności różnych narzędzi i technologii, aby pomóc programistom w dokonywaniu świadomych wyborów technologicznych podczas budowy aplikacji internetowych.

Wybrane w tej pracy frameworki znajdują się w zestawieniu. Najwyżej w rankingu uplasował się .Net umieszczony na 155 miejscu z wynikiem wydajności 42,23%. Kolejnym frameworkiem okazał się NestJS znajdując się na 419 miejscu z wydajnością 10,8%. Najniżej w rankingu znalazł się Django zajmując 472 miejsce z wydajnością na poziomie 4,3%.

Od 2013 roku, kiedy zostało opublikowane pierwsze zestawienie, badane są regularnie narzędzia. Początkowo badany był czas serializacji danych i zapytania do bazy danych. Z czasem porównanie zaczęło być rozszerzane o kolejne testy. Badanie odbywa się regularnie tworząc wiele rund dzięki czemu jego wyniki są regularnie aktualizowane. Dzięki regularnemu publikowaniu wyników, gotowe są narzędzia do wizualizacji wyników badania. W czasie rzeczywistym można śledzić postępy badania wraz z informacją ile testów przeszło oraz ile nie udało się wykonać. Przykładowy fragment z podglądu testów został zaprezentowany na rysunku 6.



Rysunek 6. Fragment podglądu testów TechEmpower - źródło: [13].

Widoczny jest nacisk położony na przejrzystość wyników.

Rozdział 3

Materiał badawczy

3.1 Django

Django to framework do tworzenia aplikacji internetowych [6]. Został wydany w 2005 roku. Nazwa pochodzi od gitarzysty Django Reinhardta. Jest to narzędzie, które nadaje się do szybkiego tworzenia zaawansowanych aplikacji internetowych, zachowując jednocześnie wysoką jakość i bezpieczeństwo kodu. Jego popularność wynika z wielu czynników, w tym szybkości wdrażania projektów, bogatej palety wbudowanych funkcji oraz zabezpieczeń.

Jedną z głównych zalet Django jest jego zdolność do szybkiego rozwoju aplikacji od pomysłu do wdrożenia. Framework ten oferuje wiele narzędzi i gotowych rozwiązań, które ułatwiają proces tworzenia stron internetowych, od autentykacji użytkowników po obsługę treści czy generowanie map strony. Dzięki temu programiści mogą skupić się na kształtowaniu logiki aplikacji. Posiada on wbudowane moduły, które pozwalają na szybki start projektu. Są to mechanizmy obrony przed najczęstszymi atakami, takimi jak wstrzykiwanie SQL czy ataki typu cross-site scripting, czy system autentykacji użytkowników, zarządzanie kontami użytkowników oraz hasłami.

Django pozwala aplikacjom na elastyczne dostosowywanie się do zmieniających się wymagań i wzrostu liczby użytkowników. Może być stosowany zarówno w małych projektach, jak i w dużych systemach obsługujących duży ruch internetowy, co czyni go uniwersalnym narzędziem dla różnorodnych zastosowań.

Kod źródłowy frameworka Django jest udostępniony publicznie na zasadach otwartego oprogramowania.

Strony takie jak Pinterest, czy Instagram korzystają z frameworku Django do swojego działania.

3.2 .NET

Framework .NET rozwijany przez firmę Microsoft od 2002 roku, stanowi kompleksową platformę programistyczną, która umożliwia tworzenie różnorodnych aplikacji komputerowych [1]. Jest to zintegrowane środowisko programistyczne zawierające narzędzia, biblioteki oraz moduły wykonawcze, które

ułatwiają proces tworzenia oprogramowania. Zapewnia infrastrukturę do uruchamiania, rozwijania i zarządzania aplikacjami na platformie Windows oraz innych systemach operacyjnych.

Jedną z istotnych cech .NET Framework jest jego wieloplatformowość, co oznacza możliwość pisania kodu raz i uruchamiania go na różnych systemach operacyjnych, takich jak Windows, Linux czy macOS. Taka elastyczność sprawia, że framework ten jest atrakcyjny dla programistów oraz firm poszukujących rozwiązania umożliwiającego tworzenie aplikacji na różnych platformach. W skład .NET Framework wchodzi rozbudowany zestaw bibliotek i narzędzi, które usprawniają proces tworzenia oprogramowania. Obejmuje on między innymi biblioteki do obsługi interfejsu użytkownika, zarządzania pamięcią, komunikacją sieciową oraz dostępem do danych.

.NET Framework jest zintegrowany z innymi technologiami Microsoftu, takimi jak Visual Studio czy platforma chmurowa Azure. Umożliwia to programistom korzystanie z kompleksowych narzędzi do tworzenia, testowania i wdrażania aplikacji, a także skalowania ich w chmurze.

3.3 NestJS

NestJS jest frameworkiem przeznaczonym do tworzenia wydajnych, skalowalnych aplikacji serwerowych opartych na Node.js [7]. Bazuje na progresywnym JavaScript, wspiera w pełni TypeScript, pozwalając jednocześnie programistom pisać w czystym JavaScript.

Framework NestJS wykorzystuje renomowane platformy HTTP Server, takie jak Express (domyślnie) i Fastify (opcjonalnie). Daje to programistom swobodę wyboru między nimi, jednocześnie oferując abstrakcję ponad nimi oraz dostęp do ich interfejsów API.

Filozofia frameworka NestJS opiera się na potrzebie stworzenia spójnej struktury aplikacji, które umożliwiają łatwe testowanie, skalowalność, luźne powiązania oraz łatwe utrzymanie kodu. Inspiracją dla tej architektury były koncepcje stosowane w frameworku Angular.

NestJS jest projektem o otwartym źródle na licencji MIT, rozwijanym od 2017 roku.

Firmy takie jak Adidas, czy Decathlon korzystają z frameworka NestJS [9].

3.4 k6

Grafana k6 to narzędzie do testowania obciążenia aplikacji internetowych oraz wykonywania testów wydajnościowych [5]. Jest to część ekosystemu Grafana, znanej platformy do monitorowania i analizy danych, co zapewnia użytkownikom możliwość integracji testów wydajnościowych z analizą danych i wizualizacją wyników.

Jedną z kluczowych cech narzędzia Grafana k6 jest jego zdolność do symulowania zachowania użytkowników poprzez wysyłanie zapytań HTTP i analizowanie odpowiedzi serwera. Można tworzyć zaawansowane scenariusze testowe, które odwzorowują różne zachowania użytkowników na stronie internetowej, takie jak logowanie, przeglądanie stron, czy też dodawanie produktów do koszyka. Oferuje ono również bogate możliwości konfiguracyjne, które pozwalają dostosować testy do różnych

scenariuszy. Można określić warunki obciążeniowe, definiować progi wydajnościowe oraz zbierać szczegółowe dane diagnostyczne, które pomagają zidentyfikować przyczyny ewentualnych problemów.

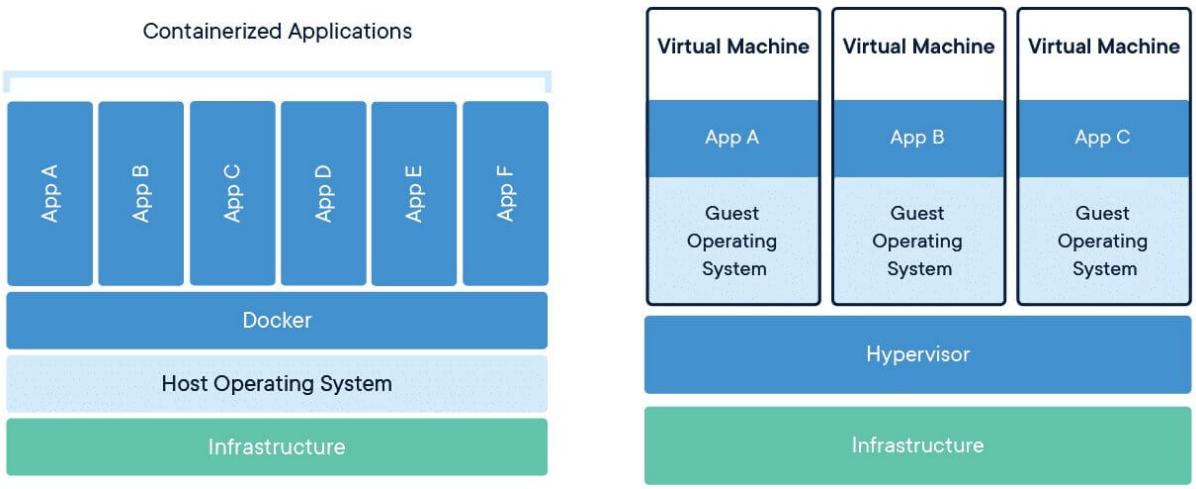
Kod źródłowy k6 jest dostępny publicznie, co oznacza, że jest dostępny dla szerokiej społeczności deweloperów i testerów. Dzięki temu można korzystać z bogatej dokumentacji, zgłaszać błędy oraz współpracować nad rozwojem narzędzia w ramach społeczności.

Grafana k6 to wszechstronne i potężne narzędzie do testowania wydajności aplikacji internetowych, które pozwala użytkownikom na symulowanie różnych scenariuszy obciążeniowych oraz monitorowanie wydajności. Dzięki temu deweloperzy i testerzy mogą zapewnić, że ich aplikacje są wydajne i odpowiadają na oczekiwania użytkowników.

3.5 Docker

Docker to platforma wirtualizacyjna, opublikowana przez firmę Docker, Inc. w marcu 2013 roku [8], która umożliwia tworzenie, uruchamianie oraz zarządzanie kontenerami aplikacji [10]. Kontenery są izolowanymi jednostkami oprogramowania, które zawierają wszystkie niezbędne zależności oraz środowisko uruchomieniowe potrzebne do poprawnego działania aplikacji. Docker wykorzystuje technologię konteneryzacji, co pozwala na przenośność aplikacji między różnymi środowiskami oraz zapewnia spójność działania na różnych platformach.

Jedną z kluczowych cech Dockera jest jego szybkość oraz lekkość. Kontenery są uruchamiane na poziomie systemu operacyjnego gospodarza, co eliminuje narzut związany z wirtualizacją tradycyjną. Porównanie architektury maszyny wirtualnej oraz Docker zostało zaprezentowane na rysunku 7.



Rysunek 7. Schemat porównania koncepcji maszyny wirtualnej oraz Dockera - źródło [3]

Dzięki temu aplikacje uruchomione w kontenerach Docker są bardzo wydajne i mogą być łatwo skalowane w zależności od obciążenia. Maszyna wirtualna dockerowa nie ma narzutu systemowego na sobie, stąd więcej takich maszyn można uruchomić naraz.

Docker jest również popularnym narzędziem w środowiskach deweloperskich oraz produkcyjnych. Pozwala on programistom tworzyć odizolowane środowiska programistyczne, co ułatwia testowanie i debugowanie aplikacji. Jest powszechnie stosowany w procesie wdrażania aplikacji, umożliwiając szybkie i spójne wdrożenie aplikacji na różne serwery oraz platformy chmurowe. Dzięki wsparciu dla narzędzi takich jak Kubernetes czy Docker Swarm, Docker jest używany do budowy i zarządzania dużymi klastrami kontenerów, co umożliwia elastyczne i skalowalne wdrażanie aplikacji w różnych środowiskach produkcyjnych.

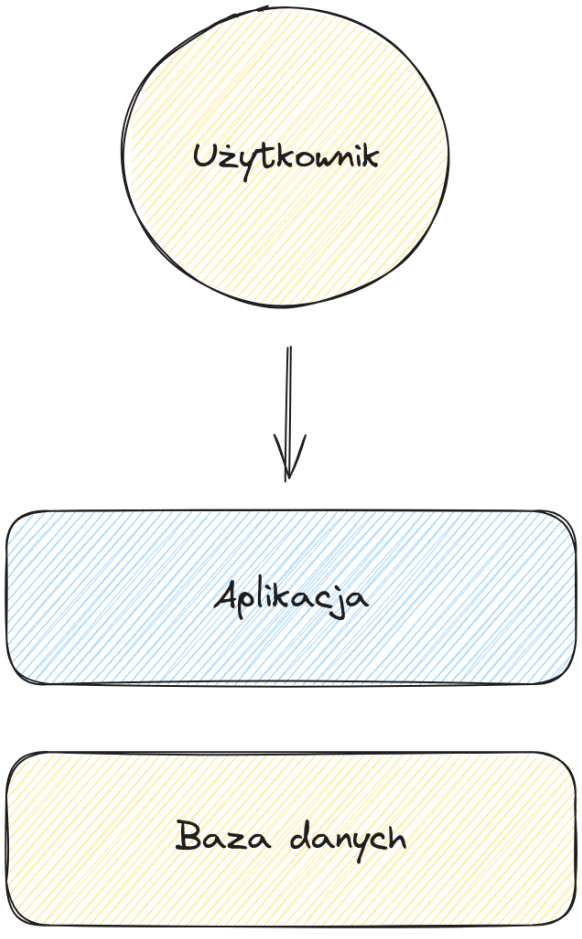
3.6 Architektura horyzontalna

Coraz częściej aplikacje są sprzedawane jako serwisy dostępne w chmurze [12]. Pozwala to na skalowanie rozwiązań w zależności od zapotrzebowania. Proste aplikacje pisze się w architekturze wertykalnej, co oznacza, że istnieje jedna instancja aplikacji, która wykonuje wymagane operacje. W przypadku wzrostu zapotrzebowania na moc obliczeniową, można zaopatrzyć się w maszynę spełniającą oczekiwane wymagania. Schemat architektury wertykalnej został zaprezentowany na rysunku 8. Takie rozwiązanie jest mało elastyczne na wahania potrzebnej mocy obliczeniowej. Podmiana instancji wymaga zatrzymania chociaż na chwilę działania usługi. Utrzymywanie urządzenia o dużej mocy obliczeniowej, tylko po to wytrzymało potencjalne skoki zapotrzebowania na moc obliczeniową wiąże się ze zwiększonymi kosztami.

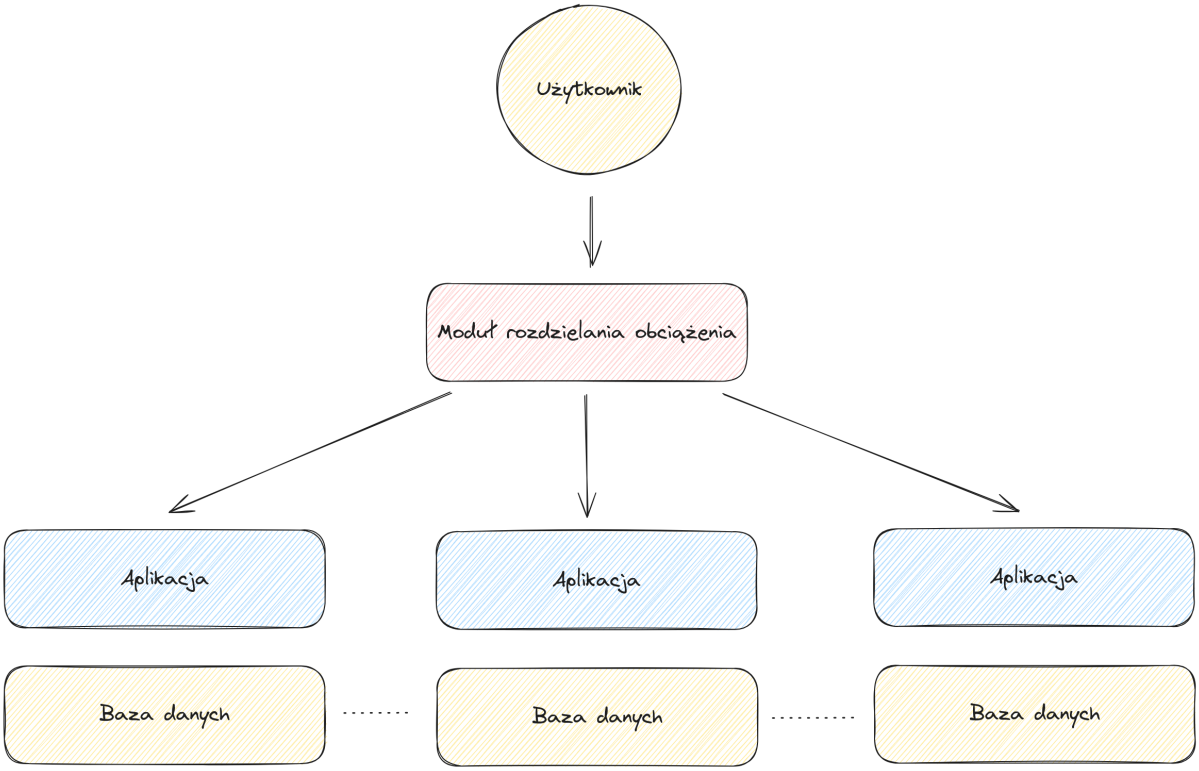
Aby rozwiązać ten problem stosuje się architekturę horyzontalną. Schemat architektury horyzontalnej został zaprezentowany na rysunku 9. Pośrednikiem komunikacji użytkownika z aplikacją jest urządzenie które przydziela instancję aplikacji użytkownikowi, tak, że z perspektywy użytkownika stale widoczna jest jedna aplikacja, natomiast może być uruchomionych wiele instancji w zależności od zapotrzebowania. Kiedy ruch jest bardzo mały, tylko jedna aplikacji może być uruchomiona. Wraz ze wzrostem obciążenia mogą być dynamicznie uruchamiane kolejne instancje aplikacji. Jest to możliwe dzięki środowiskom chmurowym, gdzie uruchomienie kolejnej instancji aplikacji nie wymaga postawienia kolejnego urządzenia fizycznego.

3.7 ORM

ORM (ang. Object-Relational Mapping) to a więc warstwa łącząca system bazy danych z aplikacją [15]. Taka warstwa automatycznie mapuje obiekty bazy z obiektami biznesowymi. Często ORM jest powiązany z konkretnym frameworkiem. Można pominąć system ORM i pisać surowe zapytania do bazy, natomiast utrzymanie takiej aplikacji jest znacznie trudniejsze. ORM zapewnia nam kompatybilność zapytań z dowolnym wspieranym silnikiem bazodanowym.



Rysunek 8. Schemat architektury wertykalnej



Rysunek 9. Schemat architektury horyzontalnej

Rozdział 4

Metody badawcze

Ważnym aspektem badań naukowych jest efektywne zarządzanie czasem, koniecznym do osiągnięcia pożądanych wyników. Często precyzyjność pomiarów nie jest decydująca, lecz wystarczająco dokładna wartość przybliżona. Dlatego też, przed przystąpieniem do dłuższych badań, często przeprowadza się szybkie testy w celu wstępnego oszacowania oczekiwanych rezultatów. Ten proces pozwala na lepsze zrozumienie wyników oraz skuteczne wykorzystanie zasobów badawczych. W rezultacie, efektywność czasowa staje się kluczowym elementem strategii badawczej, prowadząc do bardziej skutecznych i wydajnych działań naukowych.

Stąd przy większych rozwiązaniach, w kontekście niniejszej pracy, stosowana jest horyzontalna architektura pozwalająca rozłożyć większy ruch użytkowników. W badaniach została zastosowana jedna jednostka aplikacji. Należy założyć, że obsługa większej liczby użytkowników może być łatwo uzyskana poprzez zwielokrotnienie uruchomionych instancji. Każdy z badanych frameworków jest przygotowany do obsługi architektury horyzontalnej.

4.1 Badanie pojedynczego zapytania

Badania zostały rozpoczęte od analizy pojedynczego zapytania w wybranych frameworkach. Głównym celem tego eksperymentu było zbadanie czasu potrzebnego do uzyskania danych z bazy danych. Warto zaznaczyć, że każdy z analizowanych frameworków operował na tych samych zbiorach danych, co umożliwiło porównanie szybkości odpowiedzi na zapytanie między badanymi narzędziami.

Wyniki tego badania pozwalają na początkową weryfikację efektywności poszczególnych frameworków w obsłudze pojedynczych zapytań, co jest kluczowe przy projektowaniu aplikacji. Analiza czasu odpowiedzi na zapytania umożliwia identyfikację potencjalnych obszarów do poprawy oraz wybranie obszarów potrzebujących usprawnienia. Jest to pierwszy krok w celu porównania efektywności narzędzi pomocny w wyborze optymalnej technologii w zależności od konkretnych potrzeb projektowych. Wnioski z tego badania mają istotne znaczenie dla procesu decyzyjnego związanego z wyborem odpowiednich narzędzi i technologii.

4.2 Badania limitu użytkowników

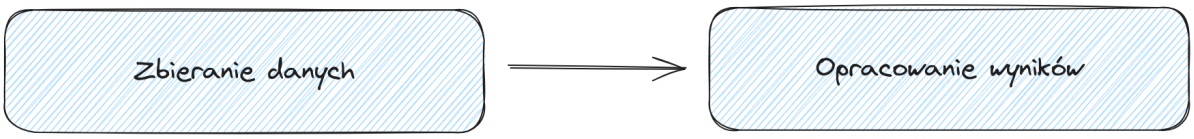
Kolejnym etapem badań było zbadanie granic liczby użytkowników, których wybrane narzędzia mogą obsłużyć równolegle. Za kryterium niepowodzenia przyjęto sytuacje, w których narzędzie zwracało błąd lub nie udawało się uzyskać odpowiedzi w określonym czasie. W celu określenia tych granic, przeprowadzono serię testów z różnymi liczbami użytkowników, wykorzystując metodę wyszukiwania binarnego. Warto podkreślić, że uzyskane wyniki są przybliżone, ponieważ mogą nieco się różnić w zależności od warunków testowych. Analiza granic jest kluczowa dla oceny skalowalności i wydajności badanych narzędzi w kontekście obsługi większej liczby użytkowników. Otrzymane rezultaty pozwalają na lepsze zrozumienie możliwości i ograniczeń poszczególnych frameworków, co przyczynia się do bardziej świadomego wyboru technologii w projektach wymagających obsługi wielu użytkowników jednocześnie.

Analiza granic liczby użytkowników obsługiwanych równolegle przez badane narzędzia jest istotna z perspektywy projektowania i skalowania systemów, zwłaszcza w kontekście aplikacji o dużej liczbie użytkowników. Poznanie tych granic pozwala na określenie optymalnej konfiguracji środowiska oraz planowanie zasobów potrzebnych do obsługi oczekiwanej liczby użytkowników. Identyfikacja punktów granicznych umożliwia programistom dostosowanie strategii zarządzania obciążeniem oraz wprowadzenie optymalizacji w celu poprawy wydajności systemu.

Rozdział 5

Przygotowanie aplikacji

W celu przeprowadzenia badania nad wydajnością oraz porównaniem trzech różnych aplikacji serwujących podobne API, zdecydowano się na implementację trzech aplikacji w różnych technologiach: Django, .NET oraz NestJS. Każda z aplikacji została skonfigurowana do korzystania z bazy danych PostgreSQL. Schemat środowiska badawczego został zaprezentowany na rysunku 11. Całe środowisko, a więc aplikacja oraz baza danych, zostało uruchomione w kontenerach Docker w lokalnym środowisku. Część zbierania danych została oddzielona od samego opracowywania wyników. Grafana K6 zapisywał pliki z wynikami, które następnie były konsumowane przez skrypt opracowujący wyniki. Skrypty zostały napisane w Jupiter Notebook, które pozwala uruchamiać skrypty systemowe oraz w języku python w ramach arkusza. Zaletą tego rozwiązania jest zapisany nie tylko skrypt ale również wynik standardowego wyjścia programu. Dowolny fragment arkusz może zostać uruchomiony powtórnie, co pozwala na szybkie korygowanie skryptów gdy otrzymamy nieoczekiwany wynik. Jest to również dogodne środowisko do przeglądania danych oraz ich rysowania. Sekwencja badań została zaprezentowana na rysunku 10.



Rysunek 10. Sekwencja badań

Po zakończeniu implementacji każdej z nich, przygotowano kilka scenariuszy testowych, służących do oceny wydajności każdej z aplikacji. Do przeprowadzenia testów wydajnościowych wykorzystano narzędzie Grafana k6, które umożliwiło monitorowanie i symulację obciążenia aplikacji. Sam framework Grafana K6 był uruchamiany jako krok w ramach skryptu, tak by uruchomić program z oczekiwanymi w danym teście parametrami. W niektórych testach potrzebne było wielokrotne uruchomienie scenariusza testowego.

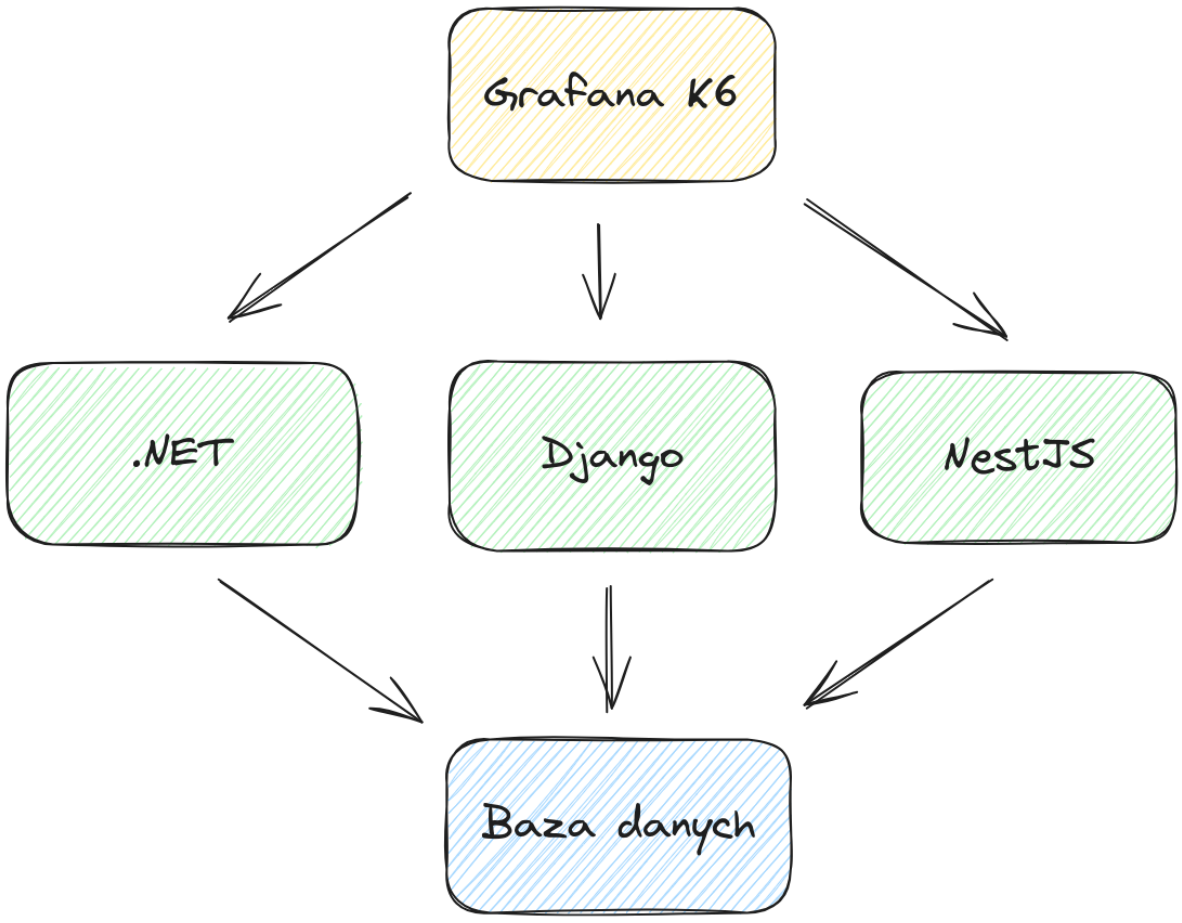
Testowane wersje narzędzi zostały przedstawione w tabeli 1. Wyszczególniona została konkretna wersja każdego z użytych frameworków oraz środowisko w którym aplikacja została uruchomiona.

Framework	Środowisko uruchomieniowe	ORM	Baza danych
NestJS 10	Node 18	TypeORM 0.3.17	PostgreSQL
ASP.NET Core 7	.NET 7.0	EF Core 7	PostgreSQL
Django 4.2	Python 3.11	Django ORM 4.2	PostgreSQL

Tablica 1. Wersje badanych narzędzi

5.1 Zbiory danych

W celu przeprowadzenia badania konieczne było przygotowanie odpowiednich zbiorów danych, które miały posłużyć do symulacji różnych scenariuszy. W tym kontekście przygotowano dwa zbiory danych, aby umożliwić różnorodne analizy:



Rysunek 11. Schemat środowiska badawczego

- **FWB_0** - Jest to zbiór pusty, pozbawiony jakichkolwiek elementów. Brak danych w tym zbiorze ma posłużyć do sprawdzenia zachowania systemu w sytuacji, gdy nie ma żadnych rekordów do przetworzenia.
- **FWB_100K** - Ten zbiór składa się z 100 000 elementów. Każdy element tego zbioru reprezentuje pojedynczy rekord w bazie danych i zawiera unikalne identyfikatory ID (numery) oraz nazwy (tekstowe). Zbiór ten został przygotowany w celu przetestowania wydajności systemu oraz jego reakcji na duże ilości danych.

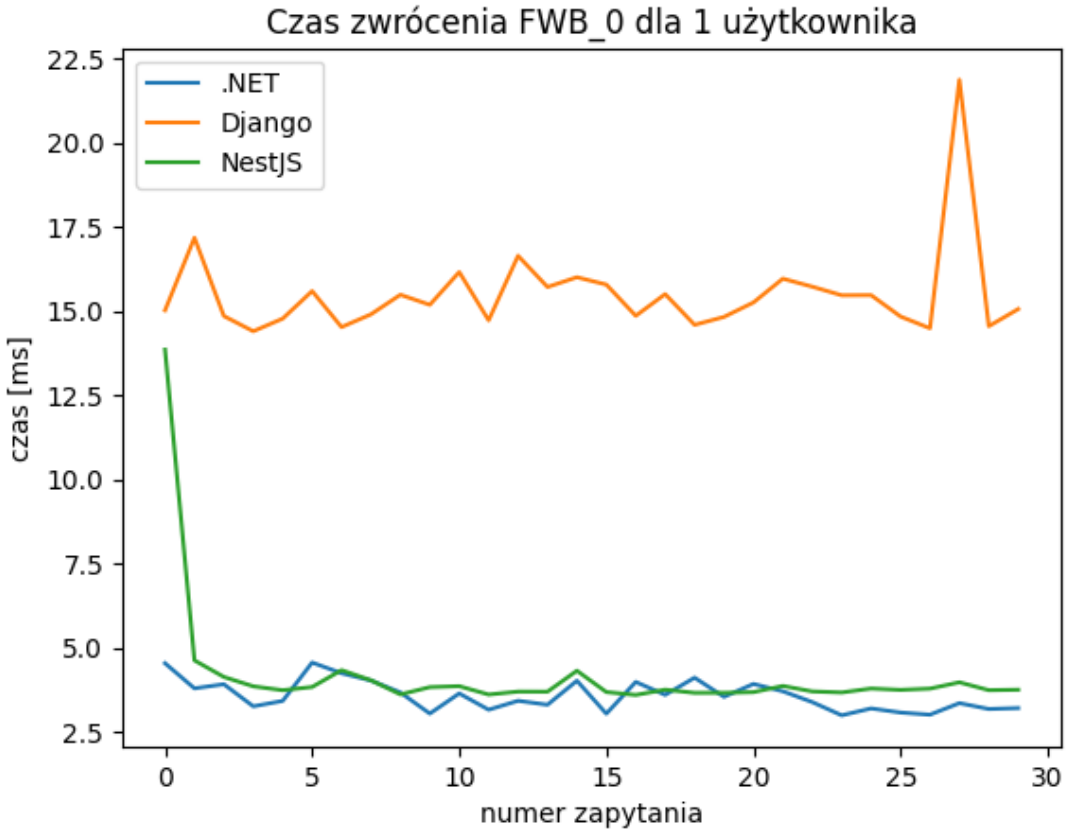
Przygotowanie tych zbiorów danych stanowiło niezbędny krok przed przystąpieniem do właściwej analizy i symulacji różnych scenariuszy w badaniu. Istotą badania systemu w sytuacji gdy nie ma on żadnych wartości do zwrócenia jest istniejący narzut potrzebny do przetworzenia zapytania. W takim wypadku nie istnieje potrzeba konstruowania kompleksowych modeli do zwrócenia. Wynik jest prosty, a więc badane jest stałe obciążenie pojawiające się w zapytaniu dowolnej wielkości.

Jako drugi zbiór został wybrany zbiór z dużą ilością danych. W tym wypadku stały narzut związany z przetwarzaniem jest również obecny natomiast poprzez dużą ilość danych powinien mieć marginalne znaczenie. Każdy framework w momencie tworzenia miał inne priorytety, stąd dla jednego duża ilość może być przygniatająca, podczas gdy inny może przetworzyć większe ilości.

Rozdział 6

Wyniki i dyskusja

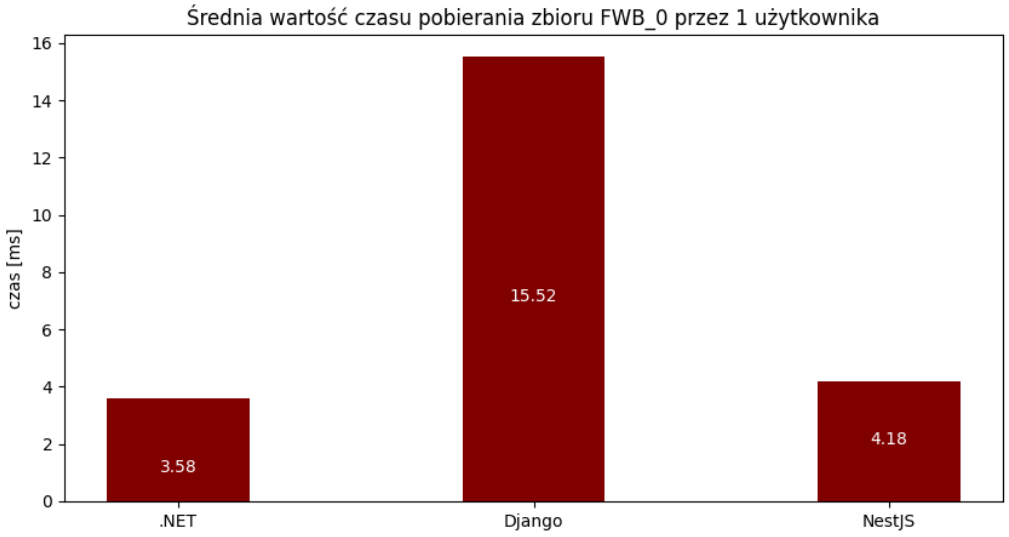
6.1 Szeregowe zapytania dla zbioru FWB_0



Rysunek 12. Czas zwrócenia zbioru FWB_0 dla 1 użytkownika

Zmierzone wartości otrzymane podczas badania zostały zobrazowane na rysunku 12. Prezentuje on czas zwrócenia wartości przez framework w mierzonym oknie czasowym. Standardowe odchylenie

dla czasu zmierzonego dla frameworka .NET wyniosło 0,45 ms. Było to najmniejsze wahanie na tle innych badanych narzędzi. Dla Django standardowe odchylenie to 1,37 ms. Trochę większe odchylenie uzyskał NestJS uzyskując wartość 1,84 ms. Widoczne jest większe odchylenie podczas pierwszego zapytania dla NestJS. Późniejsze zapytania dostają odpowiedź zdecydowanie szybciej.



Rysunek 13. Średni czas zwrócenia zbioru FWB_0 dla 1 użytkownika

Średni czas zwrócenia odpowiedzi podczas tego badania został zaprezentowany na rysunku 13. Zdecydowanie najdłuższy odpowiedzi przypadł Django osiągając wynik 15,52 ms. Około 3 razy mniejszy czas uzyskał NestJS. Wygranym tego zestawienia został .NET uzyskując najszybszy średni czas odpowiedzi na poziomie 3.58 ms.

6.2 Szeregowe zapytania dla zbioru FWB_100K

Zestawienie pokazujące czas zwrócenia zbioru FWB_100K dla badanych frameworków zostało zaprezentowane na rysunku 14. Widoczna jest ponad 2 razy mniejsza liczba zapytań wykonanych w takim samym oknie czasowym dla zbioru FWB_0.

Najniższe standardowe odchylenie dla frameworka .NET wyniosło 86,20 ms. Nieco większe odchylenie uzyskał NestJS na poziomie 114,36 ms. Dla Django odchylenie standardowe uplasowało się na poziomie 150,83.

Średni czas odpowiedzi na zapytanie zwracające zbiór FWB_100K został zaprezentowany na rysunku 15. Najdłuższy czas odpowiedzi uzyskał Django na poziomie 4,26 s. Ponad 3 razy krótszy czas przypadł NestJS z wynikiem 1,27 s. Najszybszym wynikiem wykazał się .NET z czasem 1,04 s.

6.3 Limity równoległych zapytań dla zbioru FWB_0

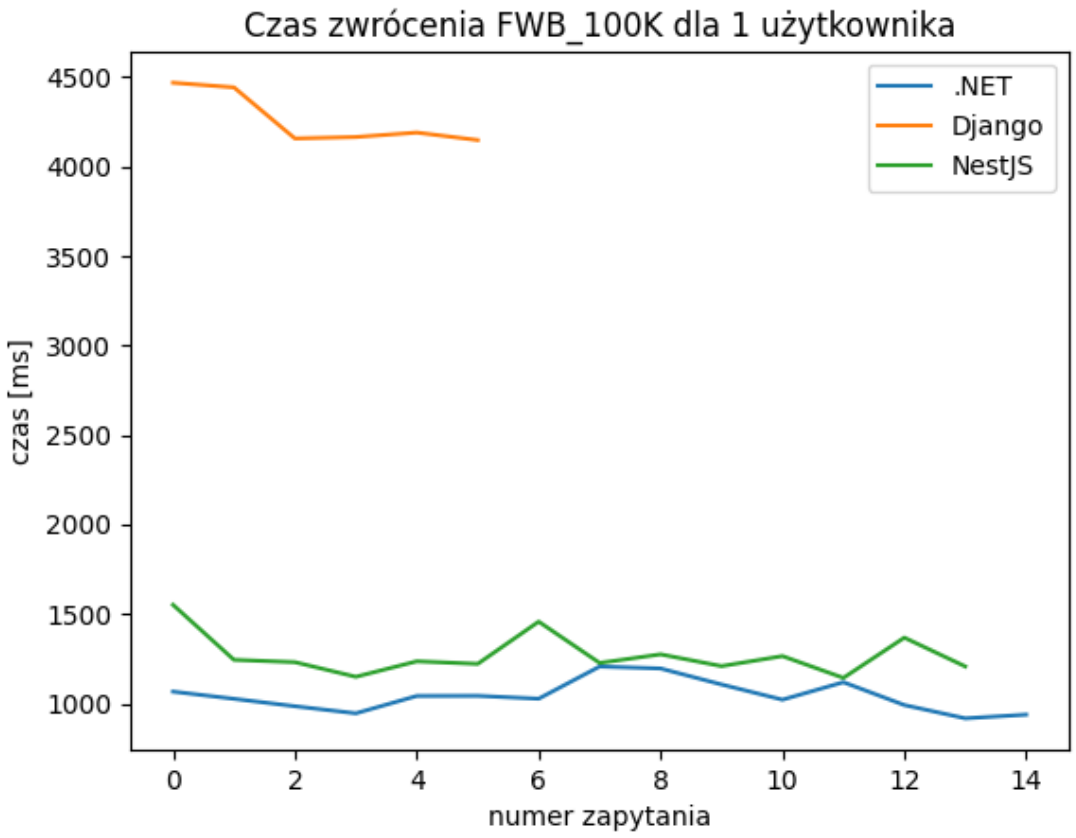
Wyniki badania limitu równoległych zapytań zostały zaprezentowane na rysunku 16.

Zestawienie zostało zdominowane przez NestJS, które uzyskało wynik na poziomie ponad 131 000 użytkowników. Kolejny na podium uplasował się .NET z 13-krotnie niższym limitem na poziomie 10 000. 3-krotnie niższy od .NET limit przypadł Django z wynikiem na poziomie 3 000.

6.4 Limity równoległych zapytań dla zbioru FWB_100K

Wyniki badania limitów zostały zaprezentowane na rysunku 17.

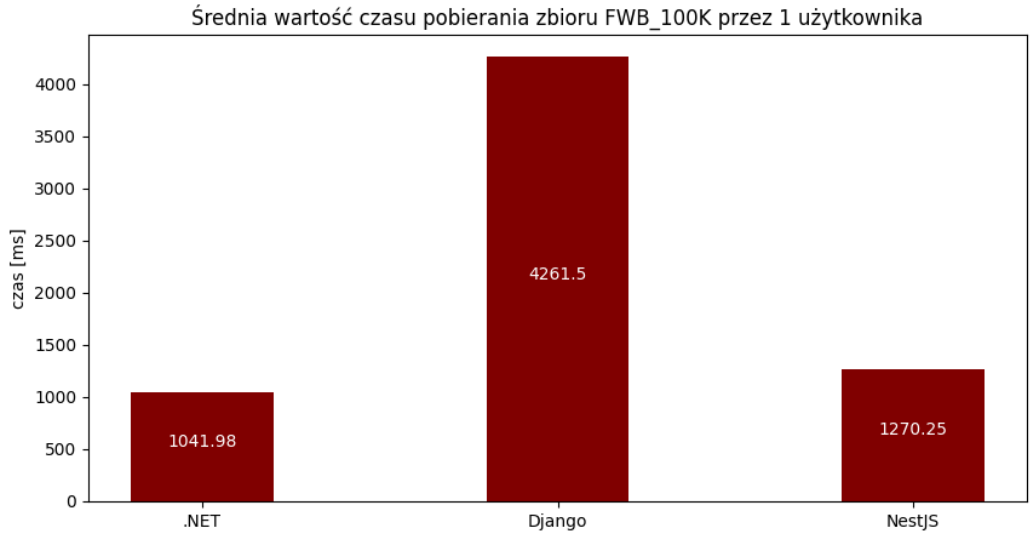
Widoczna jest zmiana zależności wyników pomiędzy frameworkami. Największą liczbę użytkowników, na poziomie 2 tysięcy, uzyskał Django. 12-krotnie mniejszą liczbę użytkowników obsłużył NestJS. .NET obsłużył 2 krotnie mniejszą liczbę użytkowników niż NestJS.



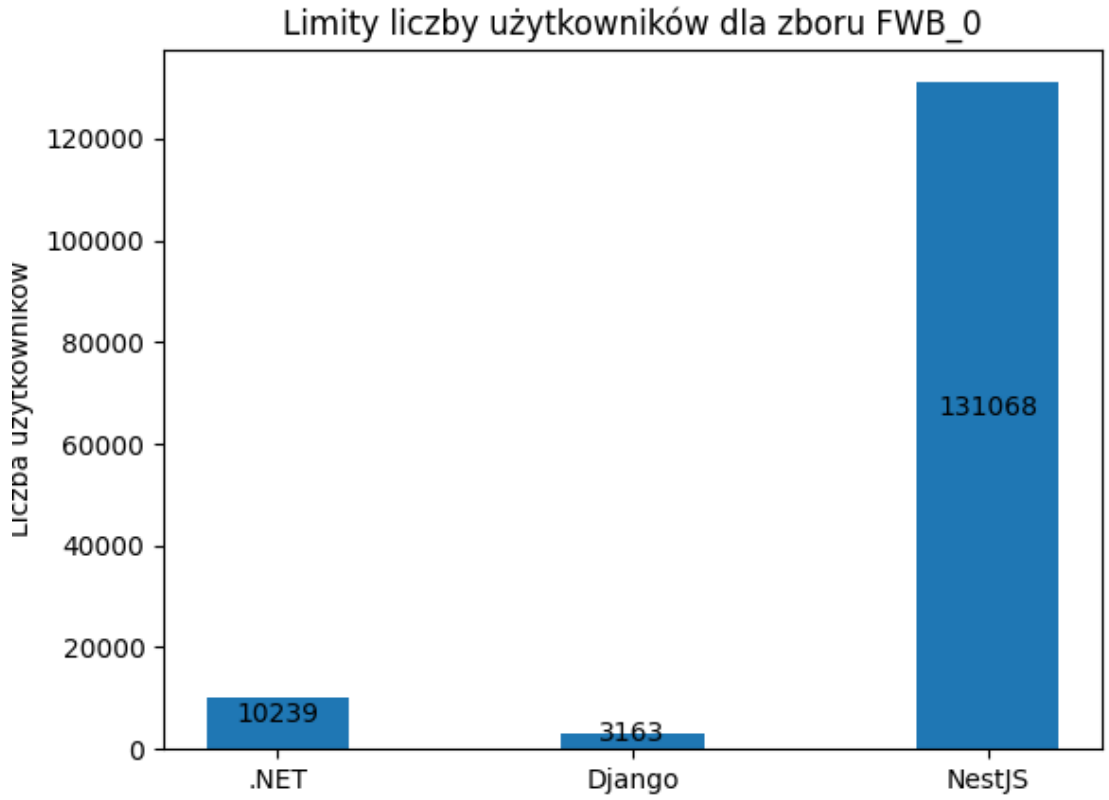
Rysunek 14. Czas zwrócenia zbioru FWB_100K listy dla 1 użytkownika

6.5 Pliki źródłowe

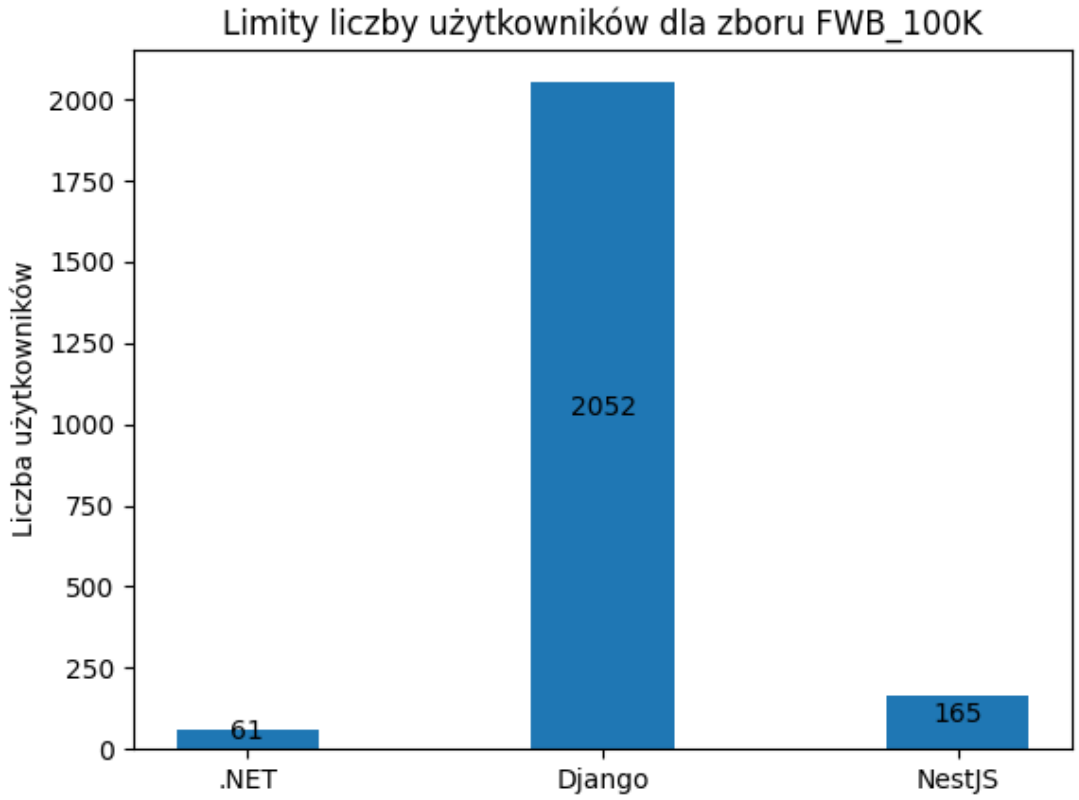
Repozytorium z implementacją aplikacji oraz narzędzi służacych przy badaniu znajduje się pod adresem `https://github.com/lukomski/framework_benchmark`. Dzięki podanym plikom możliwe jest samodzielne odtworzenie badanych środowisk i zweryfikowanie otrzymanych wyników.



Rysunek 15. Średni czas zwrócenia zbioru FWB_100K dla 1 użytkownika



Rysunek 16. Limit liczby użytkowników przy zwracaniu zbioru FWB_0



Rysunek 17. Limit liczby użytkowników przy zwracaniu zbioru FWB_100K

Rozdział 7

Podsumowanie i wnioski

Badanie szeregowych zapytań zarówno dla zbioru FWB_0 jak i FWB_100K wskazało na Django jako framework wykazujący się kilku lub kilkunastokrotnie dłuższym czasem odpowiedzi. Ponadto podczas bania limitu użytkowników wykazał się dużą odpornością na wzrost liczby użytkowników przy obsłudze większego zbioru danych.

W testach równoległych zapytań NestJS oraz .NET wypadają na podobnym poziomie z konsekwentną przewagą .NET. NestJS wykazuje się kilkunastokrotnie wyższym limitem obsługiwanych użytkowników dla zwracanego zbioru FWB_0. Dla zbioru FWB_100K nadal limit jest dwukrotnie wyższy od limitu .NET.

Nie da się wyodrębnić jednoznacznie zwycięzcę zestawienia. Każdy framework wykazał się mocnymi stronami na tle innych w różnych kontekstach.

Wybór rozwiązania w projekcie zależy od wielu czynników, wśród których badane parametry są jednymi z wielu. Niezwykle istotny jest czynnik ludzki w projekcie. Osoby w projekcie kierują się różnymi motywacjami oraz mają doświadczenie z przeróżnymi narzędziami. Zbadane obszary rzucają światło na fragment złożonego procesu decyzyjnego dotyczącego wyboru frameworka.

Bibliografia

[1] *.NET | Build. Test. Deploy. — dotnet.microsoft.com*, https://dotnet.microsoft.com/, [Accessed 17-03-2024].

[2] *A 2024 benchmark of main Web API frameworks — blog.okami101.io*, https://blog.okami101.io/2023/12/a-2024-benchmark-of-main-web-api-frameworks/, [Accessed 17-04-2024].

[3] *A Complete Overview of Docker Architecture | Cherry Servers — cherryservers.com*, https://www.cherryservers.com/blog/a-complete-overview-of-docker-architecture, [Accessed 27-05-2024].

[4] Atlassian, *Różne rodzaje testowania oprogramowania | Atlassian — atlassian.com*, https://www.atlassian.com/pl/continuous-delivery/software-testing/types-of-software-testing, [Accessed 22-05-2024].

[5] *Browser Module Documentation — k6.io*, https://k6.io/docs/using-k6-browser/overview/, [Accessed 17-03-2024].

[6] *Django overview — djangoproject.com*, https://www.djangoproject.com/start/overview/, [Accessed 17-03-2024].

[7] *Documentation | NestJS - A progressive Node.js framework — docs.nestjs.com*, https://docs.nestjs.com/, [Accessed 17-03-2024].

[8] *dotCloud - About — web.archive.org*, https://web.archive.org/web/20140702231323/https://www.dotcloud.com/about.html, [Accessed 17-03-2024].

[9] *Here's How to Hire NestJS Developers — revelo.com*, https://www.revelo.com/blog/hire-nestjs-developers, [Accessed 17-03-2024].

[10] *Home — docs.docker.com*, https://docs.docker.com/, [Accessed 17-03-2024].

[11] *Piramida testów i ciągła integracja — testerzy.pl*, https://testerzy.pl/baza-wiedzy/artykuly/piramida-testow-i-ciagla-integracja, [Accessed 22-05-2024].

[12] Ralph Mietzner, F. L. i Unger, T., „Horizontal and vertical combination of multi-tenancy patterns in service-oriented applications," *Enterprise Information Systems*, t. 5, nr. 1, s. 59–77, 2011. DOI: 10.1080/17517575.2010.492950. eprint: https://doi.org/10.1080/17517575.2010.492950. adr.: https://doi.org/10.1080/17517575.2010.492950.

[13] *TechEmpower Web Framework Performance Comparison — techempower.com*, https://www.techempower.com/benchmarks/#hw=ph&test=query&section=data-r22, [Accessed 17-04-2024].

[14] *Testowanie wydajności. Teoria — testerzy.pl*, https://testerzy.pl/baza-wiedzy/artykuly/testowanie-wydajnosci-teoria, [Accessed 27-05-2024].

[15] Xia, C., Yu, G. i Tang, M., „Efficient Implement of ORM (Object/Relational Mapping) Use in J2EE Framework: Hibernate," s. 1–3, 2009. DOI: 10.1109/CISE.2009.5365905.

Spis rysunków

Spis załączników

Załącznik 1

Algorytm wyszukiwania limitu wirtualnych użytkowników

```
    import shutil
    from datetime import datetime
    import math
    import utils
    import k6Utils
    import pandas as pd

    def test_vus_limit(
            result_base_dir: str,
            test_result_dir_prefix: str,
            script_path: str,
            setup_path: str,
            app: dict,
            inital_vus: int,
            out_file_name: str,
            end: int | None = None
            ):
        # prepare directory structure
        if not os.path.exists(result_base_dir):
            os.makedirs(result_base_dir)

        dt_string = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
        print("date and time =", dt_string)

        dir_name = f'./{result_base_dir}/{test_result_dir_prefix}{
dt_string}'
        print(dir_name)
        if not os.path.exists(dir_name):
            os.makedirs(dir_name)

        # copy script
        shutil.copy(script_path, dir_name)
        shutil.copy(setup_path, dir_name)

        out_file = open(out_file_name, "a")
        out_file.write(f'\n-------------------------')
        out_file.close()

        vus = inital_vus
```

```
39          beg = 1
40          end = end
41          threshold = 0
42          while (True):
43              # run test
44              k6Utils.run_k6(
45                  app=app,
46                  script_path=script_path,
47                  vus=vus,
48                  dir_name=dir_name
49                  )
50
51              # load dataframe
52              dir = utils.get_last_result_dir(test_result_base_dir=
    result_base_dir)
53              path = f'./{dir}/{app["name"]}.csv'
54              df = pd.read_csv(path)
55
56              # calculate metric
57              incorrect_part = utils.get_incorrect_part(df)
58
59              print(f'Incorrect part for vus = {vus}: {incorrect_part}\n')
60
61              out_file = open(out_file_name, "a")
62              out_file.write(f'\nvus = {vus}, beg = {beg}, end = {end},
    incorrect_part = {incorrect_part}')
63              out_file.close()
64
65              if (math.isnan(incorrect_part) or incorrect_part > threshold
    ):
66                  end = vus
67                  vus = math.floor((end + beg) / 2)
68              else:
69                  beg = vus
70                  vus = vus * 2 if end == None else math.floor((end + beg)
     / 2)
71
72              if end and end - beg < 2:
73                  break
```

Załącznik 2

Konfiguracja docker compose

```
services:
  django:
    build:
      context: ..
      dockerfile: ./docker/django/Dockerfile.prod
    command: gunicorn root.wsgi:application --bind 0.0.0.0:8000
    expose:
      - 8000
    volumes:
      - static_volume:/home/app/web/static
    depends_on:
      - db

  nginx:
    build:
      context: ..
      dockerfile: ./docker/nginx/Dockerfile.prod
    volumes:
      - static_volume:/home/app/web/staticfiles
    ports:
      - 8000:80
    depends_on:
      - django

  dotnet:
    build:
      context: ..
      dockerfile: ./docker/dotnet/Dockerfile.prod
    ports:
      - 5029:80
    depends_on:
      - db
    environment:
      ASPNETCORE_ENVIRONMENT: Development

  nestjs:
    build:
      context: ..
      dockerfile: ./docker/nestjs/Dockerfile.prod
    environment:
      - PORT=\${PORT}
```

```
42          - DB_HOST=db
43          - DB_PORT=5432
44        ports:
45          - 3000:3000
46        depends_on:
47          - db
48
49      db:
50        container_name: fwbm_db
51        build:
52            context: ..
53            dockerfile: ./docker/postgres/Dockerfile
54        restart: unless-stopped
55        volumes:
56            - ../volumes/dbStorage:/var/lib/postgresql/data
57        environment:
58            POSTGRES_DB: "fwbm"
59            POSTGRES_HOST_AUTH_METHOD: "trust"
60        ports:
61            - 5435:5432
62
63
64    volumes:
65      static_volume:
66
```